Efficiently Scheduling Task Dataflow Parallelism: A Comparison Between Swan and QUARK

Efficiently Scheduling Task Dataflow Parallelism: A Comparison Between Swan and QUARK

Hans Vandierendonck
Queen's University Belfast
United Kingdom
h.vandierendonck@qub.ac.uk

ABSTRACT

Increased system variability and irregularity of parallelism in applications put increasing demands on the efficiency of dynamic task schedulers. This paper presents a new design for a work-stealing scheduler supporting both Cilk-style recursively parallel code and parallelism deduced from dataflow dependences. Initial evaluation on a set of linear algebra kernels demonstrates that our scheduler outperforms PLASMA's QUARK scheduler by up to 12% on a 16-thread Intel Xeon and by up to 50% on a 32-thread AMD Bulldozer.

1. INTRODUCTION

The many-core roadmap for processors dictates that the number of cores on a processor chip increases at an exponential rate. Moreover, cores tend to operate at different speeds due to process variability and thermal constraints. As such, parallel task schedulers in the exascale era must make dynamic (runtime) scheduling decisions [1].

The task dataflow notation has been studied widely as a viable approach to facilitate the specification of highly parallel codes [2, 3, 4]. Task dataflow dependences specify an ordering of tasks (they leverage a task graph), which by its nature exposes a higher degree of parallelism than barrier-based models where threads wait periodically for all running tasks to complete. Dynamic schedulers are, however, prone to result in less performance than static schedulers due to runtime task scheduling overhead.

This work investigates a new design for a task dataflow dynamic scheduler. The key design goal is to minimize runtime overhead without affecting the task dataflow programming interface. The scheduler supports programs mixing recursive divide-and-conquer parallelism and task dataflow parallelism. This hybrid design simplifies, for instance, the exploitation of parallelism across multiple kernels called in succession. The scheduler combines the efficiency of Cilk's work stealing scheduler [5] for recursively parallel programs with the efficiency of the steal-half queue [6] for programs generating large numbers of simultaneously ready tasks.

We evaluate our design experimentally and compare against PLASMA's QUARK [2] scheduler on a set of level-3 BLAS kernels with irregular parallelism. In comparison to QUARK, our scheduler reduces end-to-end execution time of several linear algebra kernels by up to 12% on an Intel Xeon (Sandy Bridge) and by up to 50% on an AMD Bulldozer.

2. RELATED WORK

Several approaches to task dataflow scheduling have been experimented with. Several authors have implemented Tomasulo's algorithm in software [7, 8]. QUARK [2] is a work-stealing scheduler tuned to linear algebra problems. QUARK attempts to optimize data locality. QUARK records and enforces dependences using the starting address of a task argument. As such it is dependent on a fixed argument size. SMPSs [9] uses a comparable work stealing design with many design decisions that are similar from a high level point of view. An SMPSs extension for strided and sparse access patterns incurs a high performance penalty by scanning across all outstanding tasks when scheduling a task [10].

PARSeC/DAGuE is a distributed task scheduler [3]. It pre-computes and distributes the task graph in order to obtain low overhead scheduling.

StarPU [4] schedules tasks according to predicted task latency. Task latency is predicted using performance models selected by the programmer.

Swan [11] is a task dataflow scheduler built as an extension to Cilk [12]. As such, it fully supports nested parallelism. Contrary to other approaches, Swan attempts to keep the task graph small during execution and only expands it when necessary to discover parallelism. In the initial design, task graphs were retained centrally with the parent procedure. In this paper, a distributed storage of the task graph among the worker threads is proposed.

OpenSTREAM [13] is focused on stream parallelism. As such, the scheduler is organized the matching of producers with consumers.

XKaapi [14] employs a number of pattern-specific scheduling heuristics in order to reduce scheduling overhead. Others have similarly proposed heuristics to reduce the overhead of specific parallel patterns [15].

3. SWAN

Swan is a task based programming model that extends the Cilk language with dataflow annotations and dataflow-driven execution [11, 16]. In this language, the *spawn* keyword is inserted before a function call to indicate that the call may proceed in parallel with the continuation of the calling procedure. The *sync* keyword indicates that the execution of the procedure should be delayed until all spawned procedures have finished execution.

Figure 1 shows an example Swan program that implements matrix multiply. The various components of the programming model are explained below.

3.1 Objects

Objects are special program variables of type *versioned*

```
1 typedef float (*block_t )[16]; // 16x16 tile
2 typedef versioned<float[16][16]> vers_block_t ;
3 typedef indep<float[16][16]> in_block_t ;
4 typedef inoutdep<float[16][16]> inout_block_t ;
5
6 void mul_add(in_block_t A, in_block_t B, inout_block_t C) {
7     block_t a = (block_t)A; // Recover pointers
8     block_t b = (block_t)B; // to the raw data
9     block_t c = (block_t)C; // from the versioned objects
10    // ... serial implementation on a 16x16 tile ...
11 }
12
13 void matmul(vers_block_t * A, vers_block_t * B,
14                 vers_block_t * C, unsigned n) {
15     for( unsigned i=0; i < n; ++i ) {
16         for( unsigned j=0; j < n; ++j ) {
17             for( unsigned k=0; k < n; ++k ) {
18                 spawn mul_add( (in_block_t)A[i*n+j],
19                                ( in_block_t )B[j*n+k],
20                                ( inout_block_t )C[i*n+k] );
21             }
22         }
23     }
24     sync;
25 }
```

Figure 1: Square matrix multiplication expressed in a language supporting runtime tracking and enforcement of task dependences.

that express inter-task dependences. Objects may be passed as arguments to tasks using annotated task arguments that express the side-effects of the task on that argument. Annotated task arguments can only accept objects as arguments, not constants or generic variable types.

An object may be renamed, which means that its address is changed by the runtime system. The runtime system performs renaming to increase parallelism. The runtime system also makes sure that latent pointers to renamed objects are properly translated to the appropriate version of the object before accessing memory.

The runtime systems associates metadata to each object, e.g. to perform dependence analysis and to recover its most recent version after renaming. The runtime system stores this metadata side-by-side with the object in order to speedup the retrieval of metadata.

We further stipulate that all arguments passed to a task are unique objects. This is to avoid circular dependences of a task on itself.

3.2 Memory Usage Annotations

The arguments of spawned procedures may be annotated with *memory usage information*, i.e. how the argument is accessed by the task. The memory usage may be *input*, *output*, *input/output*, *commutative in/out* or *reduction*. An input argument is read but not written to. An output argument is written and may be read, but it is always written before it is read. Consequently, its value upon initiation of the task is irrelevant. An input/output argument (or in/out for short) may be read and written and it may be read before it is written.

A commutative in/out annotation extends the in/out semantics with the notion that consecutively spawned tasks may be executed in any order, but may not execute concurrently. Reordering is subject to the absence of other inter-task dependences. The runtime system guarantees that commutative tasks do not execute concurrently by associating a lock with each object to enforce mutual exclusion.

Our model also supports reductions, details of which have been previously published [17]. We will not discuss the support for reductions here as they pose no specific constraints for the purposes of this work.

Hyperqueues extend the programming model with queue usage annotations such as push and pop [18]. These annotations are not fundamentally different than the annotations listed above as they allow to use the same dependence tracking and scheduling techniques as discussed above.

3.3 Execution Model

The Swan execution model is an extension of the Cilk execution model. Swan behaves identical to Cilk in the absence of task arguments with memory usage annotations. The execution model differs when dataflow dependences between tasks are specified. These dataflow dependences are restricted within a procedure body. In other words, a task can depend only on a sibling, i.e., another task spawned by the parent of the first task. The dataflow dependences are furthermore determined by the order of the spawn statements in the procedure body. It is assumed that a sequential thread of execution steps through the procedure and, in the process, encounters a sequence of spawn statements. This sequence, together with the memory usage annotations, defines dependences between the spawned tasks. A dependence states that a pair of tasks *must* execute in the order that they were spawned. These tasks are added one by one to the task graph, where nodes represent dynamic task instances and edges represent task dependences.

The task graph is a directed acyclic graph (DAG) because tasks can only depend on tasks that appear before them in (serial) program order. At any moment, the roots of the DAG are tasks that are either *executing* or that are *ready to execute*. We call the list of root tasks that are ready to execute the *ready list*. It provides direct access to the ready tasks when one is needed.

Note that a Swan program may have up to one dataflow task graph per procedure body. Execution of the program may proceed by executing tasks from multiple task graphs concurrently. Swan uses random work stealing to balance execution between task graphs dynamically, depending on the degree of parallelism in each task graph.

4. SCHEDULING

The Swan scheduler is responsible for deciding what task is executed next by each worker thread. Like Cilk, the Swan scheduler is symmetric, i.e., all workers execute the same scheduling algorithm.

On encountering a spawn statement, the scheduler first checks that all dependences have been satisfied. If so, the scheduler proceeds as in the Cilk case, pursuing a work-first execution. A stack frame is pushed on the worker's deque (*double-ended queue*), which is managed like a call stack.

If dependences are not satisfied, then the task is not started for execution at this point. Instead, it creates a pending frame, a new type of frame in the Swan scheduler that represents an uninitiated task. The pending frame is inserted into the task graph that corresponds to the stack frame that

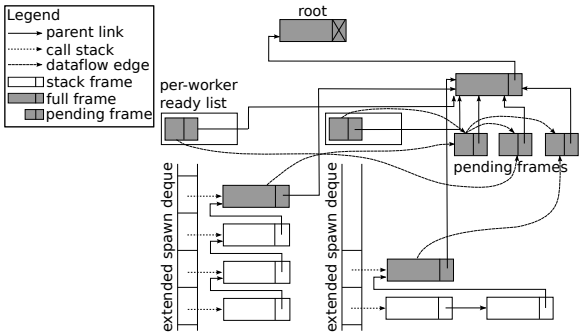


Figure 2: The Swan runtime data structures for two worker threads.

is currently at the tail of the spawn deque.[1] The scheduler then resumes execution of the stack frame at the tail of the spawn deque. Figure 2 shows the position of the ready list and the task graph in the scheduler.

The Swan scheduler applies three operations to task graphs: issue, release, and get-task. The *issue* operation registers that a task is accessing its operands (which are objects in Swan) and records the memory usage annotation. If other tasks are registered on the same objects, then the dependences are deduced and recorded, which amounts to linking the task in the task graph. If no dependences are present, then the task is executed immediately, or inserted in the ready list. The *release* operation unregisters a task, i.e., dependent tasks are notified that dependences are released and, if applicable, the dependent tasks are moved to the ready list. Finally, the *get-task* operation retrieves a runnable task from the ready list.

When a spawn deque becomes empty after completing a procedure, the scheduler first attempts to execute a ready task on the worker's ready list. This choice ensures that task graphs are kept small, as completing a task is likely to wakeup other pending tasks.

If the scheduler cannot identify ready tasks on the local ready list, it attempts a provably-good steal of the parent task. If the provably-good steal is unsuccesful, then random work stealing is attempted. Random work stealing is again designed to pick up ready pending tasks. First, a random worker is selected called the *victim*. If the victim has a non-empty ready list, then half of the tasks on the ready list are transfered to the stealing worker. This strategy minimizes work stealing [6]. One of the stolen tasks is moved to the worker's spawn deque and executed. If the vicitim's ready list is, however, empty then the scheduler tries to steal the top frame on the victim's spawn deque as in the Cilk scheduler. If all of this fails, another random victim is selected and the algorithm is repeated.

5. PLASMA INTERFACE

For the purpose of the evaluation in this paper, we tightly integrated Swan in the PLASMA system such that it can be used as a replacement of the QUARK dataflow scheduler [2]. We have implemented a number of dynamically scheduled level-3 BLAS kernels with irregular parallelism.

[1]As DAGs are restrained to a single procedure body, spawned procedures may be not ready for execution only if the parent procedure is executed in parallel, which requires it to be at the tail of the deque.

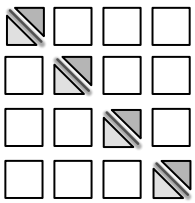


Figure 3: PLASMA matrix parts

Our implementation respects the PLASMA API. This integration enables a one-to-one comparison between Swan and QUARK as it is not affected by various implementation decisions such as data layout, library interfaces, etc.

PLASMA parallelizes level-3 BLAS kernels by decomposing them as blocked matrix operations. Hereto, matrices are decomposed in blocks, assuming an internally tuned block size. QUARK uses the starting addresses of matrix blocks to track dependences: a matrix block is shared between two tasks only if they both take the starting address of that block as an argument. Some tasks only access part of a matrix block and QUARK takes this into account. The commonly occurring parts are the lower triangular part, the diagonal and the upper triangular part of a matrix block (Figure 3). While typically only matrix blocks on the diagonal of the matrix are split in parts, it is necessary to allow per-part dependences for all matrix blocks as PLASMA supports the creation of sub-matrices which describe an arbitrary subset of the matrix. As such, the diagonal blocks on a sub-matrix may be non-diagonal blocks in the main matrix.

To integrate with PLASMA, we define the equivalent of a PLASMA descriptor (which describes the matrix layout) and PLASMA-specific dependence types that record dependences on matrix blocks (Figure 4). The Swan descriptor of a PLASMA matrix consists of the PLASMA descriptor and a 2D-array of dependence tokens. The dependence tokens consists of metadata to record actions of spawned tasks but contrary to normal variables, they do not contain data. Instead, the data is taken from the matrix. The tokens record up to three dependences to account for individual usage of the lower and upper triangular parts and the diagonal of a matrix block. The class `subobj_metadata` records such metadata and applies up to 3 times the standard Swan dependence tracking algorithm, depending on what parts of a matrix block are used by a task.

Input, output and input/output dependences can be obtained from the Swan descriptor using the `get_indep()`, `get_outdep()` and `get_inoutdep()` methods (only indep's are shown in Figure 4, other dependence types are defined similarly). These dependences hold a pointer to the corresponding token and the starting address of the matrix block's data.

Given the definition of the PLASMA matrix descriptor and dependence types in Swan, linear algebra kernels can be expressed in Swan and scheduled using task dataflow parallelism. Figure 5 shows how the `dormqr` function is declared and how it is used. `dormqr` accesses the lower-triangular part of a block A (indicated by the additional template argument `sub::lo`) and accesses blocks T and C in full. After instantiating the matrix descriptors in PLASMA and Swan formats, the appropriate matrix blocks, annotated with usage information, are obtained using the `get_`*X*`dep()` methods.

```
1  struct sub {
2     enum parts_id_t {
3        lo = 1, diag = 2, up = 4,
4        lodiag = lo | diag,
5        updiag = diag | up,
6        all = lo | diag | up };
7  };
8  template<typename T, sub::parts_id_t _parts=sub::all>
9  class plasma_indep {
10     static const sub:: parts_id_t parts = _parts;
11     subobj_metadata<sub> * meta; // dependence tracking
12     T * addr;
13
14     static plasma_indep<T,_Part>
15     create( subobj_metadata<sub> * meta, T * addr ) { ... }
16
17 public:
18     const T * get_addr() const { return addr; }
19 };
20
21 template<typename T>
22 class swan_desc {
23     PLASMA_desc desc;
24     subobj_metadata<sub> * tokens;
25
26 public:
27     swan_desc( const PLASMA_desc & _desc ) {
28        // Copy PLASMA_desc and setup 2D array of tokens
29     }
30     T * get_addr( int m, int n ) const {
31        return plasma_getaddr( desc, m, n );
32     }
33     template<sub::parts_id_t part = sub:: parts_id_t :: all>
34     plasma_indep<T,part> get_indep( int m, int n ) const {
35        return plasma_indep<T,part>::create(
36           get_token( m, n ), get_addr( m, n ) );
37     }
38 private:
39     subobj_metadata<sub> * get_token( int m, int n ) {
40        return tokens [...];
41     }
42 };
```

Figure 4: Swan interface to PLASMA descriptor

```
1  void
2  dormqr( ..., // dimensions, transforms
3          plasma_indep<double,sub::lo> A,
4          plasma_indep<double> T,
5          plasma_inoutdep<double> C );
6
7  PLASMA_desc A = ...;
8  PLASMA_desc T = ...;
9  swan_desc<double> As( A );
10 swan_desc<double> Ts( T );
11 spawn dormqr( ...,
12               As.get_indep<sub::lo>(k, k),
13               Ts.get_indep(k, k), // defaults to sub::all
14               As.get_inoutdep(k, n), ... );
```

Figure 5: Usage of Swan/PLASMA descriptor

6. EVALUATION

We compare the performance of Swan and QUARK to schedule three linear algebra kernels with irregular parallelism: Cholesky factorization, QR factorization and LU factorization with partial pivoting. Our comparison is performed on two machines: a dual-socket 2 GHz Intel Xeon Sandy Bridge E5-2650 (2x8 threads) and a dual-socket 2.1GHz AMD Opteron 6272 (2x16 threads). In the AMD processor, every pair of cores shares a floating-point unit.

We use PLASMA 2.6.0, gcc 4.9.2 and CentOS 6.5 on both machines. On Intel we use Intel MKL 11.1.2 for the basic single-threaded BLAS kernels. On AMD we use ACML 5.3.1.

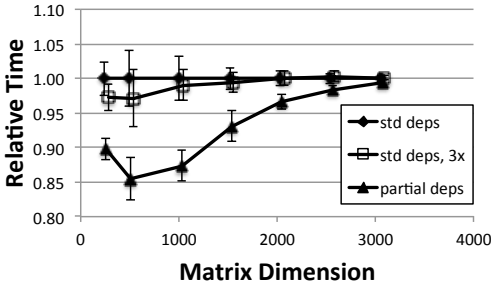


Figure 6: Overhead of tripple dependence tracking.

6.1 Dependence Tracking on Object Parts

Firstly, we validate the design of tracking dependences on object parts (one dependence chain per part of a matrix block). This analyses is performed exclusively using Swan on the Intel machine. Figure 6 compares three scenarios. The first scenario ("std deps") measures the performance of QR factorization while assuming that tasks access full matrix blocks. Only one dependence is recorded per matrix block. In the second scenario ("std deps, 3x"), we make the same assumption but we record 3 dependences per block, one for each part. The parallelism in the first two versions is identical. In the third version ("partial deps"), again three dependences are recorded per usage of a full matrix block, but the QR algorithm correctly records dependences on parts of matrix blocks. As such, the parallelism is higher in the third scenario, although dependences are recorded three times in the majority of cases. We furthermore vary the matrix dimension (the block size is kept constant to PLASMA's default of 128).

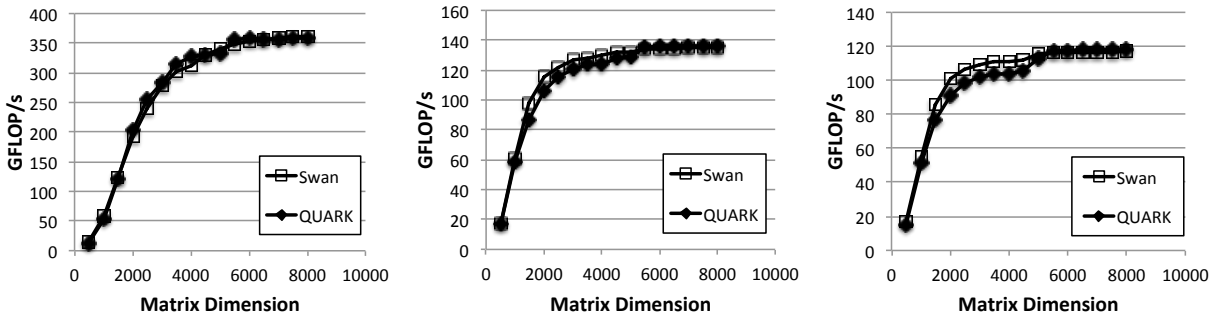
Figure 6 demonstrates that tracking dependences three times per task argument incurs little overhead. In fact, it results in a minor speedup. However the standard deviation, depicted using error bars, shows that this speedup is not statistically significant. Annotating partial usage of matrix blocks results in reduced execution time. We note this improvement for matrices with dimensions 500 to 2000, which in practice means that the degree of parallelism must be low in comparison to the block size and number of threads.

We conclude that our implementation enables increased parallelism without significant performance overhead in cases where only full matrix blocks are accessed.

6.2 Evaluation on Sandy Bridge

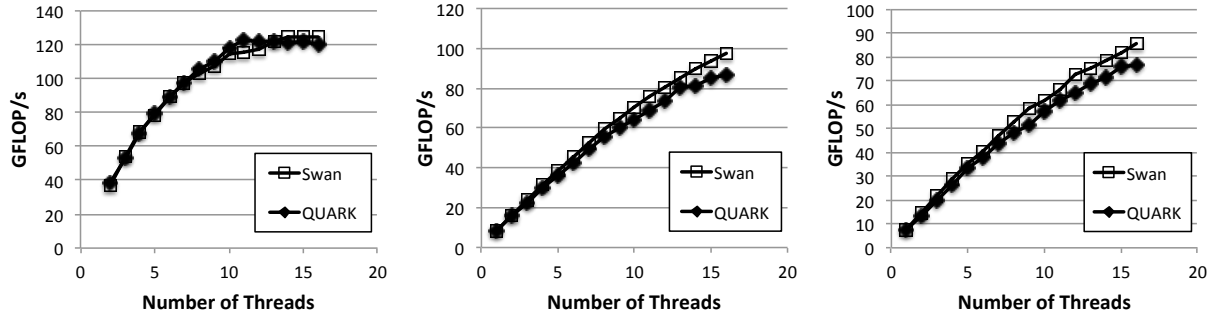
Figure 7 shows the performance of cholesky, QR and LU with partial pivoting when executing on 16 threads. Cholesky decomposition performs nearly equally with Swan and QUARK. Performance of QR, however, is between 3.8% and 12.5% faster with Swan than with QUARK for matrix dimensions up to 4000. LU is between 5.8% and 11.9% faster with Swan for matrix dimensions up to 4500.

Figure 8 shows that the performance differences grow with



(a) Cholesky (spotrf) (b) QR (dgeqrf) (c) LU (dgetrf_incpiv)

Figure 7: Performance comparison of Swan and QUARK for varying matrix dimension on Sandy Bridge using 16 threads.



(a) Cholesky (spotrf) (b) QR (dgeqrf) (c) LU (dgetrf_incpiv)

Figure 8: Performance comparison of Swan and QUARK for varying thread count and a $1500x1500$ matrix on Sandy Bridge.

an increasing thread count for a $1500x1500$ matrix. At the highest thread count, Swan executes Cholesky, QR and LU faster by 3.8%, 12.5% and 11.9%, respectively.

6.3 Evaluation on Bulldozer

Figure 9 shows the performance of the kernels when using the full Bulldozer machine (32 threads). We do not find noteworthy performance differences in this comparison for Cholesky. On LU, Swan outperforms QUARK by 5.6%–10.6% for matrix dimensions between 3000 and 5000. On QR, Swan is significantly faster, over 12% and up to 22.8% for matrix dimensions larger than 2500.

Investigating the variation with thread count (Figure 10), we see a marked difference between Swan and QUARK on the three kernels. Note that we applied thread pinning for both runtimes such that no floating-point units are shared between threads when 16 threads or less are used. Swan is able to quickly utilize most of the available performance, while performance increases more slowly after 16 threads. In contrast, QUARK needs to utilize all threads to reach close to peak performance. On 16 threads, Swan outperforms QUARK by 44–53%.

7. CONCLUSION

Swan is a versatile scheduler that has been proven in distinct scenarios, including pipeline parallelism, recursive parallelism and in this paper for linear algebra computations. The scheduler is optimized to schedule both recursive (divide-and-conquer) parallelism and task dataflow parallelism. In a one-to-one comparison with PLASMA, we demonstrate performance benefits up to 10% on a range of matrix dimensions on an Intel Sandy Bridge machine. Moreover, we demonstrate up to 22.8% performance improvement on a fully utilized AMD Bulldozer machine.

8. ACKNOWLEDGMENT

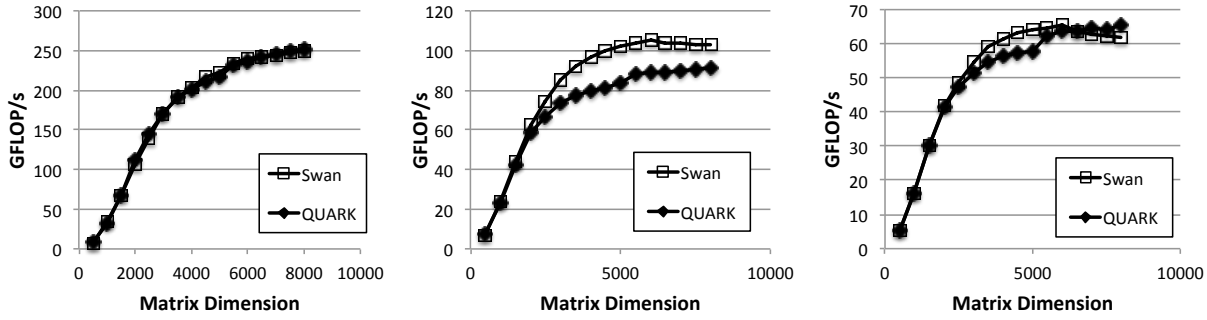
This work is partly supported by the European Community's Seventh Framework Programme (FP7/2007-2013) under the NovoSoft project (Marie Curie Actions, grant agreement 327744), the ASAP project (grant agreement 619706) and by the EPSRC under project EP/L027402/1.

9. REFERENCES

[1] J. Dongarra, P. Beckman, and et al, "The international exascale software project roadmap," *International Journal of High Performance Computer Applications*, vol. 25, no. 1, pp. 3–60, 2011.

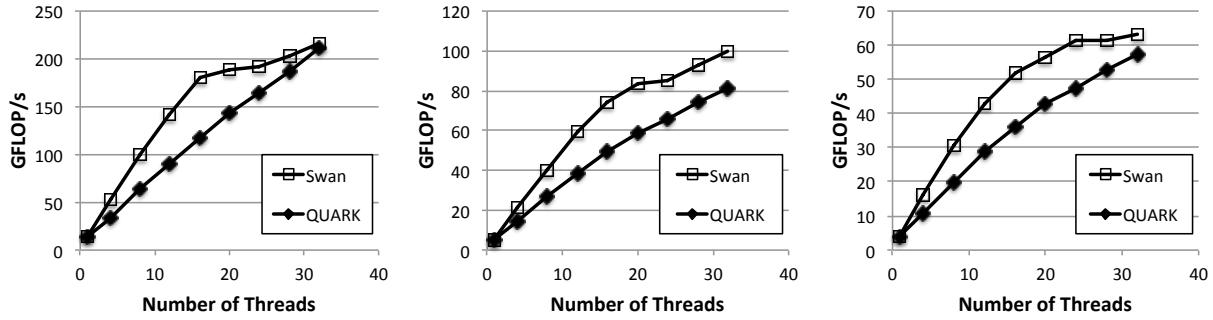
[2] J. Kurzak, P. Luszczek, and et al, *Multithreading in the PLASMA Library*. CRC Press, 2013, ch. 3, pp. 119–142.

[3] G. Bosilca, A. Bouteiller, and et al, "Dague: A generic distributed dag engine for high performance



(a) Cholesky (spotrf) (b) QR (dgeqrf) (c) LU (dgetrf_incpiv)

Figure 9: Performance comparison of Swan and QUARK for varying matrix dimension and maximum threads on Bulldozer.



(a) Cholesky (spotrf) (b) QR (dgeqrf) (c) LU (dgetrf_incpiv)

Figure 10: Performance comparison of Swan and QUARK for varying thread count and a $4500x4500$ matrix on Bulldozer.

computing," *Parallel Comput.*, vol. 38, no. 1-2, pp. 37–51, Jan. 2012.

[4] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2010.

[5] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," in *FOCS*. 1994, pp. 356–368.

[6] D. Hendler and N. Shavit, "Non-blocking steal-half work queues," in *PODC*, 2002, pp. 280–289.

[7] E. Chan, F. G. Van Zee, and et al, "Supermatrix: a multithreaded runtime scheduling system for algorithms-by-blocks," in *PPoPP*, 2008, pp. 123–132.

[8] J. Kurzak and J. Dongarra, "Fully dynamic scheduler for numerical computing on multicore processors," University of Tennessee, Tech. Rep. UT-CS-09-643, Jun. 2009, LAPACK Working Note 220.

[9] J. M. Perez, R. M. Badia, and J. Labarta, "A dependency-aware task-based programming environment for multicore architectures," in *CLUSTER*, Sep. 2008, pp. 142–151.

[10] ——, "Handling task dependencies under strided and aliased references," in *ICS*, 2010, pp. 263–274.

[11] H. Vandierendonck, G. Tzenakis, and D. S. Nikolopoulos, "A unified scheduler for recursive and task dataflow parallelism," in *PACT*, Oct. 2011, pp. 1–11.

[12] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multi-threaded language," in *PLDI*, 1998, pp. 212–223.

[13] A. Pop and A. Cohen, "OpenSTREAM: Expressiveness and data-flow compilation of OpenMP streaming programs," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 53:1–53:25, Jan. 2013.

[14] T. Gautier, F. Lementec, V. Faucher, and B. Raffin, "X-kaapi: A multi paradigm runtime for multicore architectures," in *ICPP*, 2013, pp. 728–735.

[15] A. Tzannes, G. C. Caragea, R. Barua, and U. Vishkin, "Lazy binary-splitting: a run-time adaptive work-stealing scheduler," in *PPoPP*, 2010, pp. 179–190.

[16] H. Vandierendonck, P. Pratikakis, and D. S. Nikolopoulos, "Parallel programming of general-purpose programs using task-based programming models," in *Proc. of the Workshop on Hot Topics in Parallelism*, May 2011, p. 6.

[17] H. Vandierendonck, G. Tzenakis, and D. S. Nikolopoulos, "Analysis of dependence tracking algorithms for task dataflow execution," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 4, pp. 61:1–61:24, Dec. 2013.

[18] H. Vandierendonck, K. Chronaki, and D. S. Nikolopoulos, "Deterministic scale-free pipeline parallelism with hyperqueues," in *SC*, 2013, pp. 32:1–32:12.